Brad Abrams

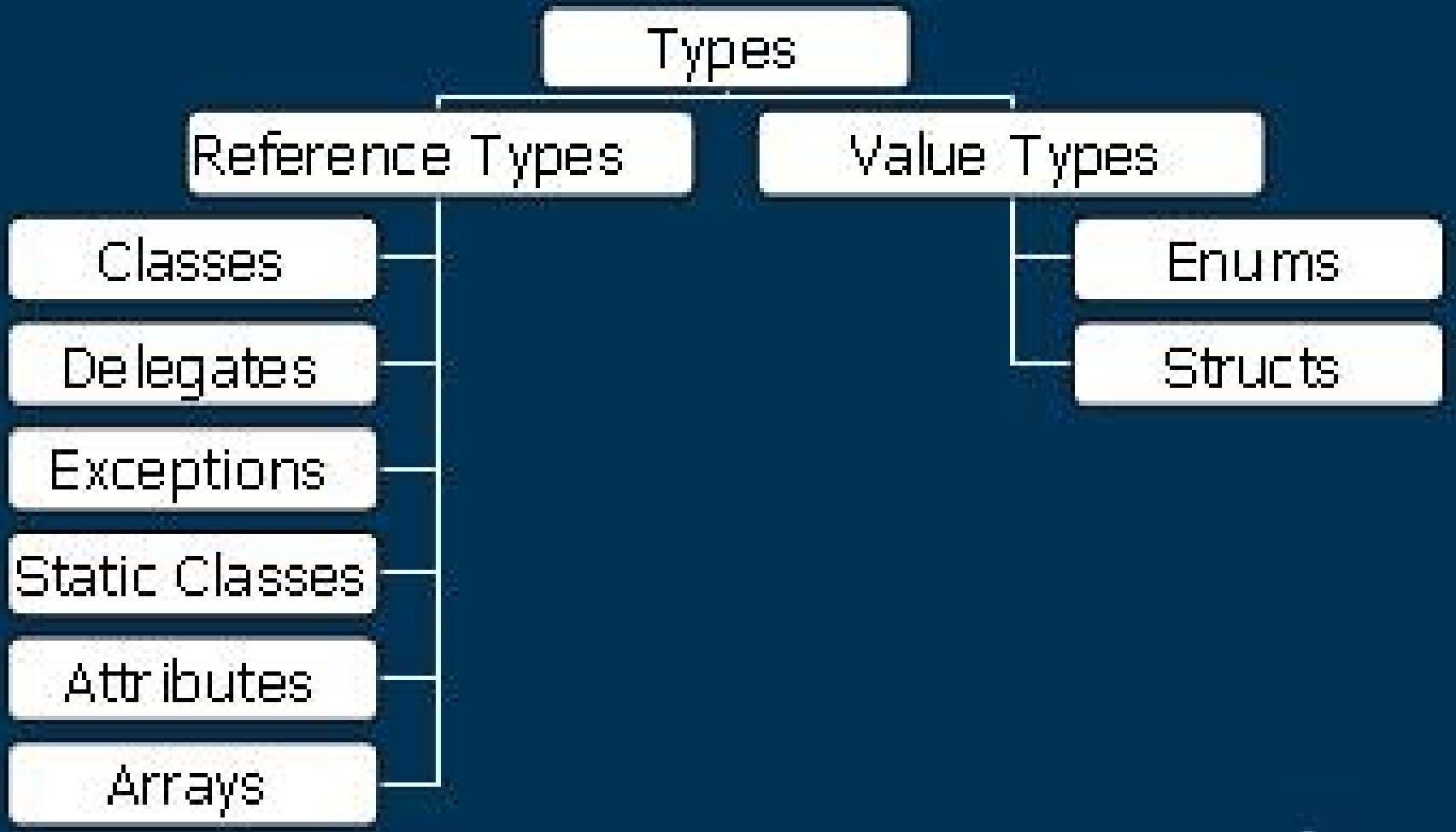
Designing Microsoft® .NET Class Libraries

June 2004

Lesson 2: A Rich Type System

- After successfully completing this lesson, you will be able to:
  - Articulate the considerations for choosing one kind of type over another
  - Correctly use each of kind of type in the system

Choosing the Right Type



Reference Types and Value Types

- Reference Types
  - Garbage collection (GC) heap allocated
  - Assignments copy the reference
  - Arrays are allocated out-of-line
    - Start out as an array of null references
  - Passed by reference
    - Actually, references to the objects are passed by value
- Value Types (structs):
  - Stack allocated
  - Assignments copy the value
  - Arrays are allocated in-line
    - Arrays are arrays of the data type
  - Passed by value

Reference Types and Value Types

- Boxing
  - Allocates box, copies value into it
- Unboxing
  - Checks type of box, CLR copies value out

```
int i = 123;
object o = i;
int j = (int) o;
```

Reference Types and Value Types (continued)

- When to use what?
  - Value types
    - Value semantics (such as an Int32)
    - Small instance size (< 16 bytes)
      - For efficient copying
    - Typically immutable

      For example, int is immutable; you can't change the 5ness of 5
  - Use reference types everywhere else

Using Structs

```
public struct Int32 : IComparable,
                      IFormattable {}
```

- When to use:
  - Act like primitive types
  - Have an instance size under 16 bytes
  - Are immutable
  - Value semantics are desirable

Using Structs: Initialization

- What happens?

```
MyStruct[] arr = new MyStruct[10];
```

- The constructor for MyStruct is not run here

- Do not provide a default constructor
- Do design for a meaningful "zero" state
- Do not depend on a constructor always being run

Using Enums

```
public enum SpecialFolder {}
```

- Very similar to C++ enums
  - All integral values are legal; not just the defined ones
  - Always scoped by type name
- Performance the same as underlying type (such as int)
- Use enums to strongly type return types, parameters, properties, *etc.*
  - Even for Booleans

Using Enums (continued)

- Use an enum instead of static constants
- Validate enum values

```
File.Open ("foo.txt", (FileMode) 42);
```

- Do not suffix enums with "Enum"
- Avoid public nested enums
- Avoid enums for user extensible values

Using Enums: Under the Covers

- Enum values default to 0..N

```
public enum Compression  {
   Zip, SuperZip, None
 }
```

```
public struct Compression : Enum {
   public int value__;
   public const Compression Zip = 0;
   public const Compression SuperZip = 1;
   public const Compression None = 2;
}
```

Using Enums: Versioning

- Always fully specify enum values; don't rely on compiler-inserted defaults
- Defining new enum members is allowed
- Be careful about where you return those new values (should not be unexpected)
- Always have a default case in switch statements

Using Enums: Flags

- Use FlagsAttribute for enums that are combinable
  - Use powers of two for enum values
  - Use plural names
    - AttributeTargets rather than AttributeTarget
  - Provide named constants for common flags

```
[Flags] public enum AttributeTargets {
    Assembly = 0x0001,
    Module   = 0x0002,
    Class    = 0x0004,
    Struct   = 0x0008,
    Type = Class | Struct,
}
```

Using Enums: Flags Continued

- Only use Flag enums when they represent a single concept
  - AttributeTargets is good example
  - The Reflection BindingFlags is an example of what **not** to do
    - Contains many different concepts in a single value (visibility, static-ness, member kind, *etc.*)
    - Should not be suffixed in "Flags"

Using Classes

```
public class Object {}
```

- Classes are by far the most used types
- Choose to use classes over any other type
- Prefer modeling classes based on concrete concepts rather than abstract concepts
  - Examples: printers, files, folders

Using Generic Types

```
public class List<T> {}
```

- Allow for parameterized types
- Conceptually similar to C++ templates
- Supported in CLR and all Microsoft and active third-party languages
- Will be supported in the Common Language Specification (CLS) in the "Longhorn" timeframe

Before Generics

```
public class List
{
   private object[] elements;
   private int count;

   public void Add(object element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public object this[int index] {
      get { return elements[index]; }
      set { elements[index] = value; }
   }

   public int Count {
      get { return count; }
   }
}
```

```
List intList = new List();

intList.Add(1);          // Argument is boxed
intList.Add(2);          // Argument is boxed
intList.Add("Three");    // Should be an error

int i = (int)intList[0]; // Cast required
```

Generics

```
public class List<T>
{
   private T[] elements;
   private int count;

   public void Add(T element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public T this[int index] {
      get { return elements[index]; }
      set { elements[index] = va
   }

   public int Count {
      get { return count; }
   }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);          // No boxing
intList.Add(2);          // No boxing
intList.Add("Three");    // Compile-time error

int i = intList[0];      // No cast required
```

Why Use Generics

- Allow for more strongly typed APIs (developers love strongly typed APIs)
  - Compile-time type checking
  - Microsoft® IntelliSense® experience is better
  - No ugly casts
  - No boxing
- Promote consistency (example: all collections are the same)
- Increase performance
- There is less code you need to write and maintain

Defining Generic APIs

- Consider using generics whenever an API returns or takes object
- Do name generic type parameters with single letters
  - Example: List<T> rather than List<ItemType>.
- Do not place generic types in special generic-only assemblies or namespaces just because the types are generic
  - Example: Don't use System.IO.Generics
- Consider providing non-generic top type

Generic APIs and Performance

- Understand usability and performance implications of exposing generic APIs
  - Compared to creating your own strongly typed wrappers, generics are always better
  - Compared to loosely typed (example: ArrayList)
    - Generics are almost always a win for reference types (code is shared)
    - But be much more careful with value types (code is not shared)
  - Most importantly, don't use generics "just because you can"
    - Think of each usage as creating a type!

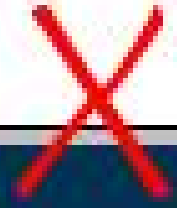
Using Generic Collections

- "Worker" Collections: Use them in internal implementation
  - List<T>, Dictionary<K,V>, Stack<T>, Queue<T>
- Object Model Collections: Use them when returning collections from APIs
  - Collection<T>, ReadOnlyCollection<T>, KeyedCollection<K,V>
- Collection Interfaces: Use them when accepting collection parameters
  - IEnumerable<T>, ICollection<T>, IList<T>, IDictionary<K,V>

Using Arrays: Mutability

- Do not return internal instances of arrays

```
public static class Path {
   private static char[] badChars = { '\"', '<'};
   public static char[] GetInvalidPathChars() {
      return badChars;
   }
}
```



Using Arrays: Mutability (continued)

- Callers can simply set the value and change your internal data structure
  - Security issue!

```
Path.GetInvalidPathChars()[0] = 'A';
```

- Clone arrays before returning them

```
return (char[]) badChars.Clone();
```

- Consider using strongly typed collections rather than arrays for ease of use

Using Arrays: "Readonly"

- Do not use readonly fields of arrays or other mutable types
  - The array reference itself is readonly and cannot be changed, but elements in the array can be changed

```
public sealed class Path {
    private Path () {}
    public static readonly char[]
            InvalidPathChars = {'\"','<','>'};
}
```

```
Path.InvalidPathChars[0] = 'A';
```

  - Callers can change the values in the array

Using Arrays: Returning Empty Instances

- Do return an empty array instead of a null reference
  - Nulls are generally unexpected to developers
  - A common usage pattern for arrays:

```
foreach (char c in t.Name) {
}
```

    - Throws an unexpected exception if Name returns null
  - Provide an Empty member if scenario is common
    - Example: String.Empty

Using Arrays: Performance

- Do not cache string or array lengths
  - The JIT hoists out the array bound checks

```
int l = a.Length;
for (i = 0; i < l; i++) {
   a[i] = 0;
}
```

```
for (i = 0; i < a.Length; i++) {
   a[i]=0;
}
```

Using Exceptions

```
public class FileNotFoundException : IOException
```

- Terminology exceptions are "thrown"
- Suffix with "Exception"
- Use exceptions rather than error codes
  - Robust: failures get noticed
- Your method is defined to do something…
  - If it succeeds in performing its purpose, return
  - If it fails to do what it was written to do, throw an exception

Using Exceptions: Continued

- Do not just map error codes on to a single exception with an error code property (example: the WMIException)
  - Use separate exception types if this is really necessary
- Do not catch and eat exceptions
  - Exceptions should be handled only where there is enough context to do the right thing
    - That is rarely true in a low-level library
  - An example of what NOT to do is File.Exists()
    - It catches all exceptions and returns false, so you have no idea what the underlying issue is

Using Exceptions: Bad Practice

- The System.Uri class did not provide an IsValid() method, so the only way to test for validity of a URI is to try to create one such as:

```
try {
   return new Uri(s, true);  // absolute
}
catch {
   // relative path
   return new Uri(Path.GetFullPath(s), true);
}
```

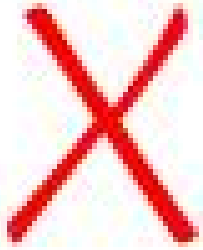
- It is likely that exceptions are used to drive control flow in about half of all cases
  - Bad performance
  - Bad programming model
  - Side note: Avoid global catch like this

Using Exceptions: Performance

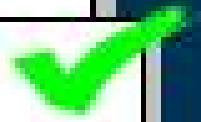
- Minimize the number of exceptions you throw in your API's success code-paths
  - You don't pay for exceptions until you throw in managed code
  - Throwing exceptions degrades performance
  - Perf counters tell you exactly how many exceptions your application is throwing
- Consider providing a way to avoid an exception being thrown

Using Exceptions: Performance (continued)

```
int i;
try {
   i = Int32.Parse("123");
}
catch (FormatException ) {
   Console.WriteLine ("Invalid");
}
```



```
int i;
if (!Int32.TryPrase ("123", out i))
{
   Console.Writeline("Invalid");
}
```



Using Exceptions: try..finally

- You should use try..finally 10 times as often as try..catch
  - Catches eat exceptions, making it hard to debug
  - Finally allows you to clean up, but lets the exception continue
  - Catch-and-rethrow has the same benefits as try..finally

Using Exceptions: Nested Exceptions

```
try {
. . .
}
catch (DivisionByZeroException e) {
    // do clean up work
    throw new BetterException (message, e);
}
```

- You may catch exceptions to re-throw them with a clearer name
- Always nest the "real" exception
- Provides same benefit as try..finally

Using Exceptions: Creating New Exceptions

- Only create separate classes if you think developers will handle the exception differently at run time

```
try {
    //some operation
}
catch (FooException fe) {
    //do some set of work
}
catch (BarException be) {
    //do some other set of work
}
```

Using Exceptions: Creating New Exceptions (continued)

- Create new exceptions when needed
  - Choose shallow and wide exception hierarchies
- Error messages:
  - Localize
  - Use a complete sentence (end in a period)
  - Don't expose privacy related information (such as file paths)

Using Exceptions: Creating New Exceptions (continued)

- Every exception should have at least the top three constructors

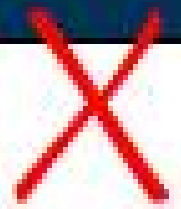
```
public class XxxException : YyyException {
    public XxxException () {}
    public XxxException (string message) {}
    public XxxException (string message,
                         Exception inner) {}
    protected XxxException (
                SerializationInfo info,
                StreamingContext context) {}
}
```

- Note: Making an exception fully serializable implies a little more work...

Using Exceptions: Bad Practice

- ArgumentNullException does not follow this pattern

```
public class ArgumentNullException :
                              ArgumentException {
 public ArgumentNullException () {}
 public ArgumentNullException (string paramName) {}
 public ArgumentNullException (string paramName,
                               string message) {}
}
```



- Justification: Argument names are much more common than message

Using Exceptions: Bad Practice

- Habit wins out and people commonly type:

```
throw new ArgumentNullException ("the value must
   pass an employee name");
```

- Rather than:

```
throw new ArgumentNullException ("Name", "the value
   must pass an employee name");
```

- We end up with odd error messages such as:

```
Unhandled Exception: System.ArgumentNullException:
Value cannot be null.
Parameter name: the value must pass an employee name
```

- Lesson: Just follow the pattern!

Using Attributes

- How do you associate information with types and members?
  - Category of a property
  - Transaction context for a method
  - XML persistence mapping
- Traditional solutions
  - Add keywords or pragmas to language
  - Use external files (examples: .IDL, .DEF)
- .NET solution: Attributes
  - A formalized notion of extensibility

Using Attributes: Example Usage

```
public class Button: Control
{
   [Category("Appearance")]
   [Description("Color of text in button")]
   [Browsable(true)]
   public Color TextColor {...}

   protected override void Paint(Graphics g) {
      TextOut(g.GetHdc(), 10, 10, "Hello");
   }

   [DllImport("gdi32", CharSet = CharSet.Auto)]
   static extern bool TextOut(int hDC,
      int x, int y, string text);
}
```

Using Attributes

```
public sealed class ObsoleteAttribute :
Attribute {}
```

- Suffix with "Attribute"
- Performance tip: Seal attribute classes if you need fast run-time lookup
- Specify the AttributeUsage attribute completely
  - Don't rely on the defaults!
  - Be as restrictive as possible
    - You can always open it up later

Using Attributes

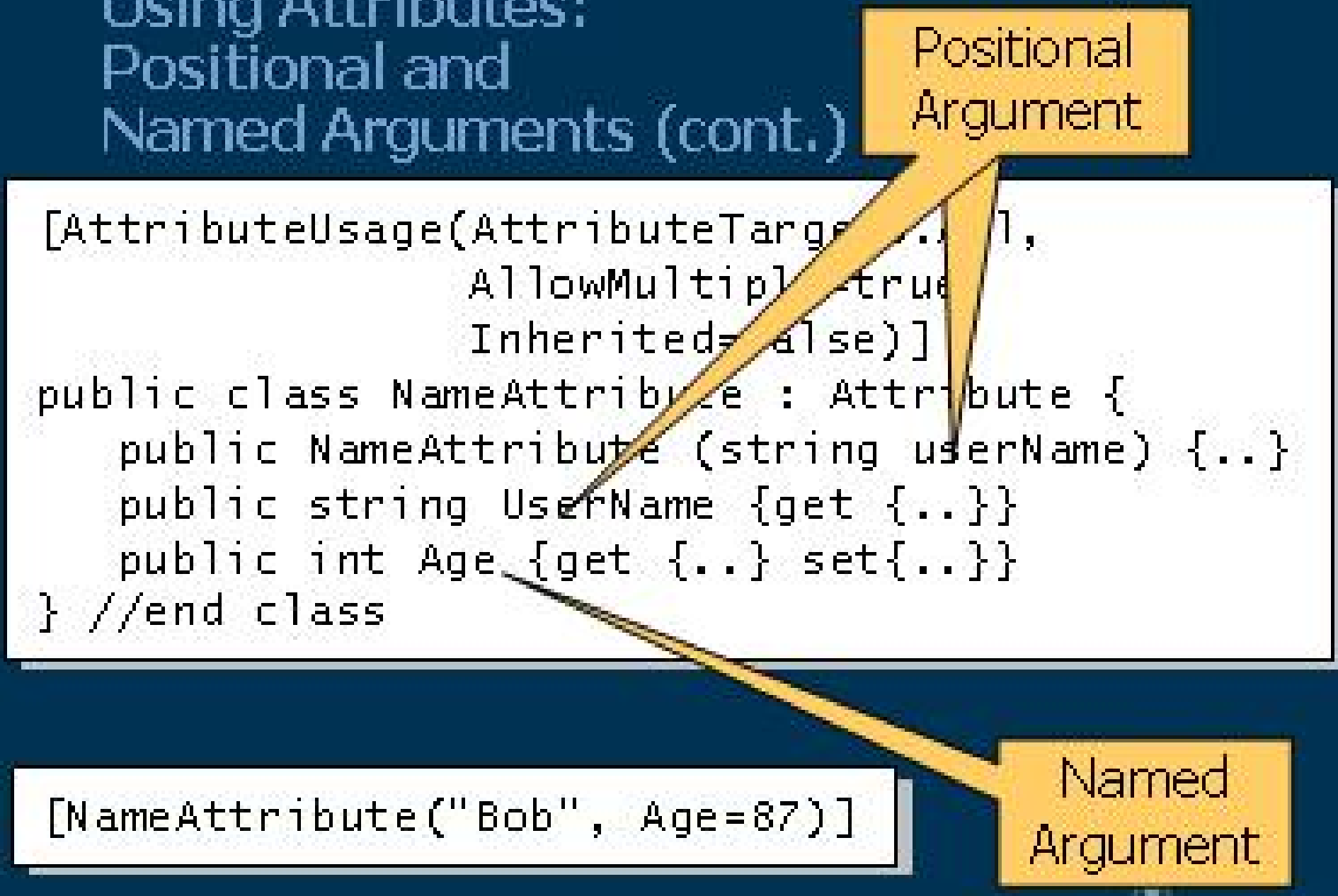
```
[AttributeUsage(
      AttributeTargets.All,
      Inherited = true,
      AllowMultiple = false)]
```

- **AttributeTargets**—Where is the attribute allowed to be applied?
- **Inherited**—Should derived members/types be considered to have this attribute?
- **AllowMultiple**—Is it legal to put more than one instance of the attribute on a particular member?

Using Attributes: Positional and Named Arguments

- Use constructor arguments for required parameters (positional arguments)
  - Provide a read-only property with the same name
- Use read-write properties for optional parameters (named arguments)
- Never use overloaded constructors

Using Attributes: Positional and Named Arguments (cont.)



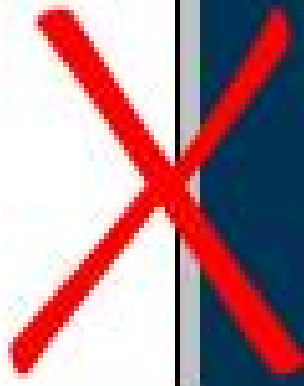
Using Attributes: Bad Design

- The "designer" slapped the CLSCompliant(false) attribute on the members just to make the warning go away
- Did not provide a CLS-compliant alternative
  - Result: Microsoft Visual Basic® users could not take advantage of this attribute

Using Attributes: Bad Design

- Not useable from Visual Basic and other CLS-complaint languages

```
[AttributeUsage(AttributeTargets.Assembly,
AllowMultiple=false, Inherited=true)]
public sealed class AssemblyFlagsAttribute :
                                        Attribute {
  [CLSCompliant(false)]
  public AssemblyFlagsAttribute(uint flags) {}

  [CLSCompliant(false)]
  public uint Flags {
     get {..}

  }
}
```



Using Attributes: Bad Design

- The fix:
  - Must provide a CLS-compliant way to set and retrieve state
  - We shipped it, so we can't break compatibility
  - Must pick a new property name since properties cannot differ only in return type

Using Attributes: Bad Design

- The fix:

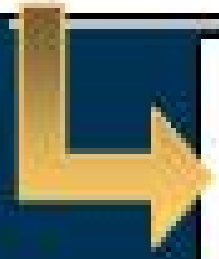
```
[AttributeUsage(AttributeTargets.Assembly,
    AllowMultiple=false, Inherited=true)]
public sealed class AssemblyFlagsAttribute :
                                        Attribute {
  [CLSCompliant(false)]
  [Obsolete("use int32 version")]
  public AssemblyFlagsAttribute(uint flags) {}

  public AssemblyFlagsAttribute(int
                                  assemblyFlags) {}

  [CLSCompliant(false)]
  [Obsolete("use AssemblyFlags property")]
  public uint Flags { get {..} }

  public int AssemblyFlags { get {..} }
}
```

Static Classes

- Static classes contain just static members
- Compromise between pure object-oriented (OO) design with usability
- Commonly used for:
  - Shortcuts for other operations (System.IO.File)
  - Functionality for which a full OO wrapper is unwarranted (System.Environment)

Static Classes: Under the Covers

```
public static class Environment {
      public static string CommandLine {
            get {..}
   }
}
```



```
public abstract sealed class Environment
{
 public static void Exit(int exitCode)
      {..}
    public static string CommandLine {
      get {..}
    }
}
```

Static Classes

- Best used when:
  - There is a clear charter for the class
    - Not a "miscellaneous" bucket
  - Not the center point of a design
    - Use sparingly
    - Watch out for disconnected design
- Static classes:
  - Are sealed
  - Have private default constructor
  - Have no instance members

Static Classes: Bad Design

- Late in the final milestone we added a method to tell if the runtime is being shut down
- However, we added the method as an instance method—making it completely uncallable

```
public sealed class Environment {
   private Environment() {} //Prevent creation
   // ---snip---
   public bool HasShutdownStarted {
      get { return nativeHasShutdown(); }
   }
   public static string UserName { get {...} }
   private static extern bool nativeHasShutdown();
   // ---snip---
}
```

Exercise: Choosing the Right Type

- Your design team wants
  - A type that represents a document
  - A type that represents a rational number
  - A way to mark types as serializable

Exercise: What Is Wrong with This Type?

```
public class DaysOfTheWeek  {
   public const int Monday = 1;
   public const int Tuesday = 2;
   public const int Wednesday = 3;
}
```

Exercise: What Is Wrong with This Type?

```
public class PrinterOnFireError :
    Exception  {

  public PrinterOnFireError (
      string printerName) {}
  public PrinterOnFireError (
      string printerName,
      string message) {}
  public PrinterOnFireError (
      string printerName,
      string message,
      Exception InnerException) {}
}
```

Exercise: What Is Wrong with This Type?

```
[Flags]
public enum Color {
  Red, Green, Blue
}
```

Exercise: What Is Wrong with This Type?

```
public class FileIoPath {
    public static string
      GetFileName(string path) {}

    public static string
      GetRootDir (string path) {}
}
```

Exercise: What Is Wrong with This Type?

```
public class CodeReviewer : Attribute  {
    public CodeReviewer (string alias) {}
    public CodeReviewer (string alias,
                         int level) {}
    public int Level {get{}; set{}}
    public string ReviewerAlias {get{}}
}
```

Lesson 2 Summary

- Use the right type for the right job
  - Value types for "primitive" types
  - Enums for named constants
  - Classes in most cases
  - Exceptions for error handling
  - Attributes for "custom keywords"
  - Static classes for utilities
  - And be careful with arrays